REMARKS

Claims 1-31 were originally presented in the subject application. Claims 1, 11 and 22 were amended, claims 2, 12 and 23 were cancelled, and claims 32-50 were added in a Preliminary Amendment dated July 23, 2003. No claims have herein been amended, added or canceled. Therefore, claims 1, 3-11, 13-22 and 24-50 remain in this case.

Request for Information

The Office Action noted a number of items listed in the specification, and incorporated by reference, requesting copies thereof for consideration. Unfortunately, only two were present in the undersigned's file and are included herewith. Applicants note that the requested item entitled, "CORBA A Guide To Common Object Request Broker Architecture," by Ron Ben-Natan, McGraw-Hill Publishers (1995), was included with Applicants' Information Disclosure Citation filed with this continuation application. We have requested and received electronic copies of 11 of the remainder of the incorporated items from the assignee and are enclosed herewith on a CD. The four remaining references will be forwarded when received from the assignee. A list of the references is appended hereto, indicating the status of each reference.

35 U.S.C. §103 Rejection

The Office Action rejected claims 1, 3-11, 13-22 and 24-50 under 35 U.S.C. §103, as allegedly obvious over Held et al. (U.S. Patent No. 5,802,367) in view of Thatte et al. (U.S. Patent No. 6,442,620). Applicants respectfully, but most strenuously, traverse this rejection.

Claim 1 recites a method of providing access to an object of a computing environment. The method comprises requesting access, by a requester, to an object located in an address space of the computing environment, the requester being resident within the address space. The method also comprises providing access to the object using a local access proxy located within the address space, wherein use of the local access proxy provides

separation of management of one or more object references to the object from management of one or more virtual memory copies of the object.

Against the requesting aspect of claim 1, for example, the Office Action cites to Held et al. at column 10, lines 62-67. According to that aspect, the requestor and the object are both located in the same address space. However, a careful reading of the cited section of Thatte et al. and the surrounding text reveals no teaching about the location of the object relative to the requestor. The cited section merely teaches that if server code corresponding to the activation request needs to execute locally (i.e., on the server node where the code is located), then the client service control manager communicates directly (presumably with the server node). The beginning of that paragraph of Held et al. at column 10, line 35 clearly indicates the context of the cited section is *remote accessing of an object*; that is, an object remote from the requestor. Applicants submit this is quite the opposite of the object being in the same address space as the requestor.

As another example, against the claim 1 aspect of use of the local access proxy providing separation of the management of object reference(s) from management of virtual memory cop(ies) of the object, the Office Action at numbered section 6 cites to Thatte et al. However, the facelets (and associated facelet-managing proxy manager) are only used when the client component application object and the server component application object are in *different* apartments (column 13, lines 42-63), which are in *separate domains* (column 11, lines 27-28). In contrast, claim 1 recites that the local access proxy is in the *same address space* as both the requestor and the object to which access is being requested.

Therefore, for at least the above reasons, Applicants submit that claim 1 cannot be obviated over Held et al. in view of Thatte et al.

Each of independent claims 11, 21 and 22 contains limitations similar to those argued above with respect to claim 1. Thus, the remarks above are equally applicable to those claims. Therefore, Applicants submit that none of claims 11, 21 or 22 can be made obvious over Held et al. in view of Thatte et al.

CONCLUSION

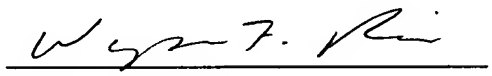
Applicants submit that the dependent claims are allowable for the same reasons as the independent claims from which they directly or ultimately depend, as well as for their additional limitations. In that regard, Applicants do not acquiesce to the alleged teachings of the cited references relative to the dependent claims.

Applicants acknowledge the references cited in the Office Action, but not substantively applied. However, Applicants expressly do not acquiesce to the alleged teachings thereof, and, in any case, submit that the pending claims are patentable thereover as well.

For all the above reasons, Applicants maintain that the claims of the subject application define patentable subject matter and earnestly request allowance of claims 1, 3-11, 13-22 and 24-50.

If a telephone conference would be of assistance in advancing prosecution of the subject application, Applicants' undersigned attorney invites the Examiner to telephone him at the number provided.

Respectfully submitted,



Wayne F. Reinke
Attorney for Applicants
Registration No.: 36,650

Dated: February 2, 2006.

HESLIN ROTHENBERG FARLEY & MESITI P.C.
5 Columbia Circle
Albany, New York 12203-5160
Telephone: (518) 452-5600
Facsimile: (518) 452-5579

APPENDIX A
STATUS OF PROVIDING COPIES OF REFERENCES AS LISTED IN THE SPECIFICATION

PAGE	TITLE	STATUS
13	"Enterprise Systems Architecture/390 Principles of Operation", IBM Publication No. SA22-7201-05, Sixth Edition (Sept. 1998)	Awaiting Hard Copy
14	CORBA A Guide To Common Object Request Broker Architecture, by Ron Ben-Natan, McGraw-Hill Publishers (1995)	Enclosed
14	Object- Oriented Programming Using SOM and DSOM, by Christina Lau, Van Nostrand Reinhold Publishers (1994)	Awaiting Hard Copy
14	"CORBA 2.2/IIOP Specification," available at WWW.OMG.ORG/library/C2INDX.HTML	Enclosed
15	"Component Broker Programming Reference Release 2.0," IBM Publication No. SC09-2810-04 (Dec. 1998)	Trying to Locate
15	"Component Broker Programming Guide Release 2.0," IBM Publication No. GO4L-2376-04 (Dec. 1998)	Trying to Locate
15	"Component Broker Advanced Programming Guide Release 2.0," IBM Publication No. SC09-2708-03 (Dec. 1998)	Trying to Locate
16	"MVS Programming: Assembler Services Guide," IBM Publication No. GC28-1762-01, Second Edition (September 1996)	Enclosed on CD Original Not Available – Updated Version Provided
16	"MVS Programming: Assembler Services Reference," IBM Publication No. GC28-1910-01, Second Edition (September 1996)	Enclosed on CD Original Not Available – Updated Version Provided
54	"OS/390 MVS Planning: Workload Management," IBM Pub. No. GC28-1761-07 (March 1999)	Enclosed on CD
54	"OS/390 MVS Programming: Workload Management Services," IBM Pub. No. GC28-1773-06 (March 1999)	Enclosed on CD Original Not Available – Updated Version Provided

APPENDIX A
STATUS OF PROVIDING COPIES OF REFERENCES AS LISTED IN THE SPECIFICATION

PAGE	TITLE	STATUS
54	"OS/390 V2R5.0 MVS Programming: Resource Recovery," IBM Publication No. GC28-1739-02(Jan. 1998)	Enclosed on CD Original Not Available – Updated Version Provided
65	"OS/390 MVS Planning: Workload Management," IBM Pub. No. GC28-1761-07 (March 1999)	Enclosed on CD Original Not Available – Updated Version Provided
65	"OS/390 MVS Programming: Workload Management Services," IBM Pub. No. GC28-1773-06 (March 1999)	Duplicate
73	"OS/390 MVS Planning: Global Resource Serialization," IBM Pub. No. GC28-1759-04 (March 1998)	Enclosed on CD Original Not Available – Updated Version Provided
73	"OS/390 MVS Programming: Authorized Assembler Services Guide," IBM Pub. No. GC28-1763-06 (March 1999)	Enclosed on CD
73	"OS/390 MVS Programming: Authorized Assembler Services Reference," IBM Pub. Nos. GC28-1764-05 (Sept. 1998)	Enclosed on CD
73	GC28-1765-07 (March 1999)	Enclosed on CD Original Not Available – Updated Version Provided
73	GC28-1766-05 (March 1999)	Enclosed on CD
73	GC28-1767-06 (Dec. 1998)	Enclosed on CD
73	"OS/390 MVS Programming: Assembler Services Guide," IBM Pub. No. GC28-1762-01 (Sept. 1996)	Duplicate
73	"OS/390 MVS Programming: Assembler Services Reference," IBM Pub. No. GC28-1910-1 (Sept. 1996)	Duplicate

CORBA

A Guide to the Common Object Request Broker Architecture

Ron Ben-Natan

McGraw-Hill

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Chapter 10

IBM SOMObjects

The SOMObjects toolkit is an IBM product that, among many other capabilities, provides a framework for developing distributed object-oriented applications based on a CORBA-compliant ORB. At the heart of SOMObjects is the System Object Model (SOM), which is the underlying technology targeted primarily at allowing the creation of *binary class libraries*. Binary class libraries are class libraries that were constructed using one object-oriented programming language and but that may be used from within another programming language. Binary class libraries allow the clients and the implementors of the classes be as decoupled as possible; the clients should not have to be recompiled if the implementation has changed internally. This type of sharing between different object-oriented development environments has been lacking, even though procedural libraries have been providing similar functionalities. SOM provides the necessary support to allow the creation of such object-oriented binary class libraries.

The basic enabler of such libraries is SOM IDL. SOM IDL is based on CORBA IDL and extends it in a number of ways necessary to support objects in the SOMObjects toolkit. The developer defines the interfaces of the objects to be included in the libraries using SOM IDL. These definitions are translated by the SOM compiler to the programming language of choice, where the methods are actually implemented. Since the interfaces are defined in SOM IDL, any programming language that has a binding for the IDL may be used for both the implementation and the invoking client. For clients, it must be possible to create and invoke requests, while for object implementations, the actual methods must be implemented. The client and the object implementation are totally decoupled; the interfaces are accessed in IDL, and so the client does not care (and does not know) what programming language was used for the object implementation.

The delivery of the requests from the client to the object implementation is done by the SOM runtime system (or the DSOM runtime system if the client and object are in different address spaces).

SOM stresses that libraries must be *binary* class libraries. This means that SOM provides an underlying runtime system that ensures binary usage of the libraries. Therefore, if an implementation change is performed that does not require a source code change to the client, no recompilation of the client is necessary. Changes to the object interface require client recompilation, since the access routines used by the client may have changed. These changes, however, are much less common; interfaces usually stabilize early in a development project's life, whereas implementations keep evolving. Changes that do not require recompilation in SOM include even such things as extending the class with more methods (actually changing the interface), adding superclasses (since SOM supports multiple inheritance), moving methods to superclasses, and even changing the size of the object. Most of these changes require client awareness in most all other object-oriented class libraries.

SOM ensures programming language neutrality through a series of design decisions. All interfaces are written in SOM IDL, and so clients can use any programming language that has a SOM IDL binding to access objects implemented in any programming language (that has a binding). This is sufficient for allowing programming language neutrality, but SOM goes a step further. SOM defines a common object model that is used by any application using SOMObjects. This object model is well defined yet is flexible enough to accommodate the object models of most object-oriented programming languages. For example, method resolution is flexibly controlled by the developer, and multiple dispatch strategies can be fitted to the appropriate programming environment or application. SOM defines a standard linkage convention allowing any programming language that can make external calls to activate the SOM runtime system and access SOM objects. This is not limited to object-oriented programming languages; in fact, the first programming language to have a SOM binding was the C programming language.

SOM includes a set of built-in libraries that make up the SOM runtime system. These classes provide the support required for the SOM object model, for method dispatching, for object reference management, and so on. The last component of the base SOM structure is the SOM compiler, which is used to translate the IDL specifications into a variety of implementation skeletons and client stubs.

SOMObjects provide much richer capabilities than are provided by SOM alone. While SOM remains the central technology of SOMObjects, and while all the added frameworks use SOM and the capabilities it provides, SOMObjects provides an extensive set of

frameworks to be used by application programmers as a very rich infrastructure basis. These frameworks include

- *Collection and communication classes.* These provide support for standard data structures and socket communication wrappers as SOM objects that can be used by application programmers.
- *The Interface Repository framework.* This provides storage and retrieval capabilities for interface information and is used by SOM and other frameworks (e.g., Distributed SOM). It can also be used by application programmers who require access to meta information.
- *Distributed SOM (DSOM).* This is a CORBA-compliant ORB implementation that extends SOM with distribution transparency.
- *The Replication Framework.* This framework allows the construction of groupware applications where multiple users view the same semantic object and participate in the manipulation of this common view. The replication framework allows objects to be associated with one another as replicas of one semantic object. The framework then provides manipulation techniques that allow the synchronization of the replicas; if one replica will be changed, the change will be propagated to all of the other replicas. These replicas exist in separate address spaces and potentially on different machines. The replication framework does not use DSOM; rather, a specialized framework is provided for specifically supporting replicated objects.
- *The Event Management Framework.* This supports event management and callback dispatch, which is necessary in single-threaded environments but is often useful even in multithreaded environments that require certain integrity guarantees for event handling.
- *The Emitter Framework.* This allows programmers to quickly and easily create compilers that read IDL definitions and produce specialized output as defined by certain templates and algorithms. These are supplied by the programmer but are inserted into the framework in predefined procedures. This process is easy to perform, allowing multiple emitters to be created. This is useful for programming language binding creation but also for such things as producing automated documentation and information for CASE tools.
- *The Persistence Framework.* This provides SOM objects with persistence capabilities.

All of these frameworks provide an initial set of classes that can immediately be used by application programmers. These frameworks are all constructed in a modular and flexible way that allows the

developer to extend any component by subclassing the default built-in class and implementing custom behavior. The frameworks are truly frameworks in the sense that multiple customizations are possible using simple replacement of different objects serving different purposes.

This chapter describes the SOMObjects Developer Toolkit, Version 2.0 as documented in IBM (1993a and 1993b). This version is supported on the AIX and OS/2 platforms and will be supported on other IBM and non-IBM operating systems including operating systems not used on personal computers or workstations (e.g., it will be supported on mainframe operating systems). The toolkit includes bindings for the C and the C++ programming languages; other programming language bindings will probably be available before this book is published. For example, the joint proposal for the Smalltalk-to-CORBA IDL binding (by HP and IBM—See Mueller, 1994a) is partly based on experience with binding SOM IDL to Smalltalk. Other programming language bindings for SOM will include non-object-oriented programming languages like COBOL.

10.1 The System Object Model (SOM)

The primary goal of SOM is to allow the production of programming language–neutral binary class libraries. SOM is the central component in the SOMObjects toolkit and provides capabilities used by all SOMObjects frameworks.

SOM separates interface specification and object implementation. Interfaces are created using the SOM IDL, while object implementations are done in programming languages which have SOM bindings. Interface definitions are also stored in the SOM Interface Repository and may be accessed as InterfaceDef objects to be used by application programs or by the SOMObjects components.

SOM is a very rich object model that can support most object models used today. It supports the notion of classes as first-class objects and metaclasses as objects describing the classes (or the classes' class, as in Smalltalk; see Goldberg and Robson, 1983). The SOM runtime system manages SOM at runtime. Class objects need to be created at runtime so that instances may be created. The class objects may then be accessed at runtime as objects, and methods can be performed for these objects. SOM supports class derivation and multiple inheritance. This allows it to support programming languages that require multiple inheritance as well as the various specifications coming out of the OMG. Many of the SOMObjects toolkit frameworks use multiple inheritance (e.g., see the use of multiple inheritance in the automatic generation of proxy classes in DSOM, discussed in Sec. 10.4).

Writing SOM applications involves the following:

- SOM IDL is used to define the interfaces and other definitions to be used.
- The SOM compiler is used to produce multiple translations of the interfaces. These will typically include header files to be used by the object's clients (implemented in one programming language), implementation skeleton files that will be used for the object implementation (which will be completed by the implementor of the object in a certain programming language), Interface Repository information, and more. The Emitter Framework (Sec. 10.7) describes how emitters can be constructed to produce additional translations of the IDL specifications.
- The client of the objects will then be developed. Include files generated by the SOM compiler will be used as if the SOM object were implemented in the client's programming language and running in the client's address space (modeled after local procedure calls). The client's program will include SOM runtime initialization code, since the management of SOM objects is done by a set of SOM components.
- The object implementation is completed based upon the skeleton produced by the SOM compiler.
- The client and implementation are compiled and linked, forming an executable. If the object implementation is changed at a later time with no changes to the interfaces, the client does not require recompilation. This is true even if the size of the object changes. When DSOM is used, the client and object implementation are not running in the same address space or necessarily on the same machine, so that the client may not even have to be halted for an object implementation to change.

10.1.1 SOM IDL

SOM IDL is based upon CORBA IDL and adheres to those definitions wherever possible. A number of extensions have been added to support SOM specific functionalities. All usages of these SOM specific IDL constructs should be delimited by

```
#ifdef
-
#endif
```

directives to allow easy location of non-CORBA-compliant constructs and automatic removal of them in later versions of SOM. For example, Implementation Statements (to be discussed below) are delimited by

```
#ifdef __SOMIDL__
...
#endif
```

directives, and private attributes or methods (not available in COBRA IDL) are delimited by

```
#ifdef __PRIVATE__
...
#endif
```

Since SOM IDL is almost identical to CORBA IDL, a full description of it is not provided here. Instead, a description of the major extensions included in SOM IDL is provided.

SOM IDL adds two type categories to CORBA IDL data types. For any SOM IDL type `T`, the type `unsigned T` is allowed in SOM IDL. CORBA IDL only allows `unsigned long` and `unsigned short`. Pointer types are also introduced by SOM IDL. For any SOM IDL type `T`, the type `T*` is allowed in SOM IDL.

The most important extension made by SOM IDL is the use of Implementation Statements. Implementation Statements specify implementation details and take the form of *instance variable declarations*, *passthru statements*, or *modifier statements*.

Instance variable declarations use syntax similar to that of ANSI-C to declare class instance variables having a SOM IDL type. These variables are private in the C++ sense; that is, they can be accessed only by the class's methods. These are different from the `__PRIVATE__` directives, which are used for private IDL methods and attributes.

Passthru statements allow blocks of code to be passed unmodified to the output files generated by the SOM compiler. A passthru statement specifies what result file the block should be copied to as well as what the code block to be copied is. Passthru statements are provided primarily for backward compatibility with SOMObjects Version 1.0.

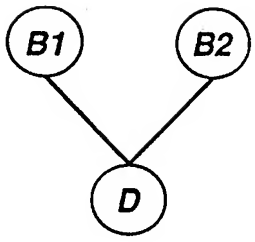
Modifier statements are used to provide additional information about IDL definitions. They provide information about interfaces, types, attributes, and methods. Table 10.1 details the most commonly used SOM modifier statements.

10.1.2 Inheritance in SOM

`SOMObject` is a class provided as part of the SOMObjects toolkit. This class defines behavior common to all SOM objects. Any class must therefore inherit (directly or indirectly) from `SOMObject`. SOM supports the notion of class inheritance, as is common in most object-oriented programming languages (e.g., in C++ as defined in Ellis and

TABLE 10.1 SOM Modifier Statements

Modifier	IDL Construct	Description
nodata	attribute	Specifies that an instance variable should not be created for the attribute. The get and set accessor methods are created for attribute access. This is used for attributes which are calculated and not stored
noset	attribute	Specifies that a set accessor methods should not be created by the SOM compiler; only a get accessor methods is created. Used for readonly (or const) attributes
persistent	attribute	Specifies that the attribute is part of the persistent representation of the type. This is used by the Persistence Framework.
callstyle=oidl	interface	Informs the SOM compiler to use old IDL style syntax.
classinit=<proc name>	interface	Informs the SOM compiler of the procedure to be used for initializing the class object. The SOM compiler will create a skeleton for the procedure implementation which will be completed by the class developer.
dllname=<filename>	interface	Specifies where the class implementation is loaded from (which library file).
functionprefix= <prefixstring>	interface	Used by the SOM compiler to prefix all methods in the class with the prefix string. This option is used to partially resolve name space conflicts.
majorversion= minorversion=	interface	Specifies the class's major and minor version numbers.
metaclass= <classname>	interface	Specifies the class's metaclass
method	method	Specifies that the method can be overridden.
procedure	method	Specifies that the method should not be overridden and informs the SOM compiler to use efficient direct dispatching for this method instead of requiring run time dispatching.
noooverride	method	Specifies that the method should not be overridden by subclasses but does not have the effect on the SOM compiler of changing the dispatching strategy.
override	method	Specifies that the method will be overridden by this class.
offset ; namelookup	method	Directs SOM as to the resolution method to be used for this method




```
interface D : B1, B2
{
    ...
};
```

Figure 10.1 A multiple inheritance example.

Stroustrup, 1990). Multiple inheritance is supported by SOM, and the class hierarchy is therefore a rooted directed acyclic graph.

Multiple inheritance introduces the possibility of potential conflicts when two superclasses define a method by the same name. SOM uses a *leftmost precedence rule*. This means that the method used will belong to the class that was leftmost in the inheritance specification. This rule is also known as the *first subclass rule*. Figure 10.1 presents an example in which the class `D` inherits from both `B1` and `B2`. Both of these base classes define a method called `f`. In SOM, `D` would inherit `f` from `B1`, since it is the first subclass (left on the superclass list in the interface clause).

SOM provides two ways by which the default conflict resolution rule can be overridden. The simplest way is to override `f` in class `D` and manually call one or both of the `f` implementations in the base classes. The other possibility is to change the ambiguity resolution rule by using a metaclass method. The default SOM ambiguity resolution rule is defined in the SOM class `SOMClass`. The programmer can override this by defining a metaclass for class `D` (instead of using the default metaclass `SOMClass`) and overriding the procedure for constructing the class's method table.

10.1.3 Metaclasses in SOM

In SOM, every class is itself an object. The class object has methods and can be used like any other object at runtime. Since the class is itself a SOM object, it must have a class—this is the metaclass (a class of a class). The class is then said to be an instance of the metaclass. The metaclass defines the methods that can be performed by the class. This includes such methods as object constructors (i.e.,

instance creation and initialization methods of the class can be added to the metaclass), methods for changing the default class hierarchy management (e.g., the resolution of ambiguities resulting from multiple inheritance), and more.

A metaclass of a class is defined using the `metaclass` modifier (see Table 10.1). A metaclass does not have to be specifically defined for every class; it may be inherited. All metaclasses inherit (directly or indirectly) from `SOMClass` (which in turn inherits from `SOMObject`, so that the metaclass hierarchy is part of the SOM object hierarchy). If no specific metaclass is defined for a class, then the default `SOMClass` metaclass will be used.

Metaclasses have their own inheritance hierarchy structure, which may differ from the class inheritance hierarchy (as opposed to Smalltalk, for instance; see Goldberg and Robson, 1983). SOM provides a unique capability regarding metaclasses that is supported by only a few object-oriented development environments. While allowing a class to freely name its metaclass and its superclasses, SOM guarantees that no method resolution errors (errors of the type "message not understood") occur for subclasses. Figure 10.2 illustrates the problem that may occur with named metaclasses and class inheritance. The code is shown in Smalltalk syntax for concreteness. In this illustration, `Account` objects have an `addOwner:` method used for adding an owner to the account. This method checks for a maximum number of owners, which is a single value for all accounts and is therefore defined by the `Account` class object. This value is supplied by the `maxOwners` method defined in `AccountClass`. The code in `addOwner:` might look like

```
addOwner: aPerson
   | max |
   max := self class maxOwners.
   ....
```

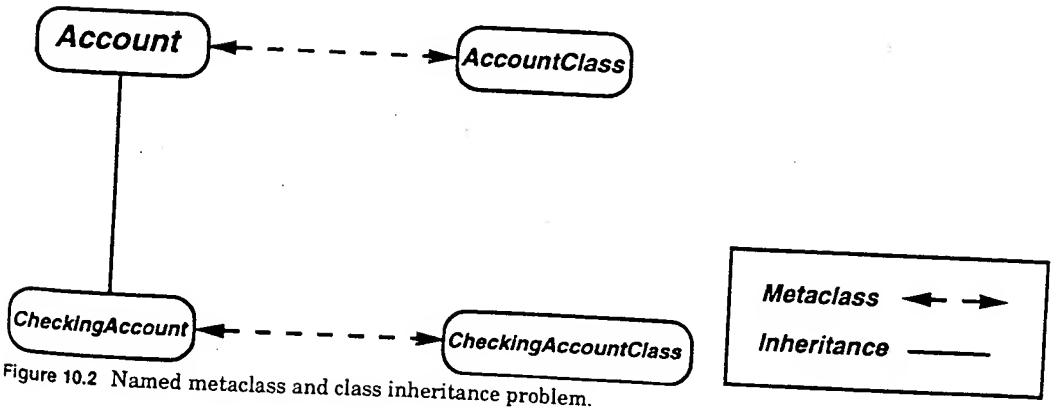


Figure 10.2 Named metaclass and class inheritance problem.

"self class maxOwners" invokes the maxOwners method on the metaclass object to retrieve the common value.

Suppose now that an instance of CheckingAccount (which names its own metaclass) were to invoke the maxOwners methods under the metaclass scenario shown by Fig. 10.2. A problem would occur, since the CheckingAccountClass metaclass does not implement the maxOwners method. When the self class maxOwners code was performed at runtime, a method resolution error would occur.

Smalltalk solves the above problem by not allowing metaclasses to be freely named by the programmer. The metaclass hierarchy matches the class hierarchy, and in our example, CheckingAccountClass would inherit from AccountClass. SOM provides a different and more flexible solution. SOM allows the metaclass to be freely named for each class. SOM will then automatically create a DerivedMetaclass, as shown in Fig. 10.3. This automatic metaclass generation will guarantee correct ambiguity resolution, since the explicitly specified metaclass is first on the inheritance list. An explicit definition of the maxOwners method in our example will therefore always be used before an inherited implementation. Derived metaclasses are also used when multiple inheritance is involved. In this case the class has multiple superclasses, so that the metaclass must inherit from multiple sources.

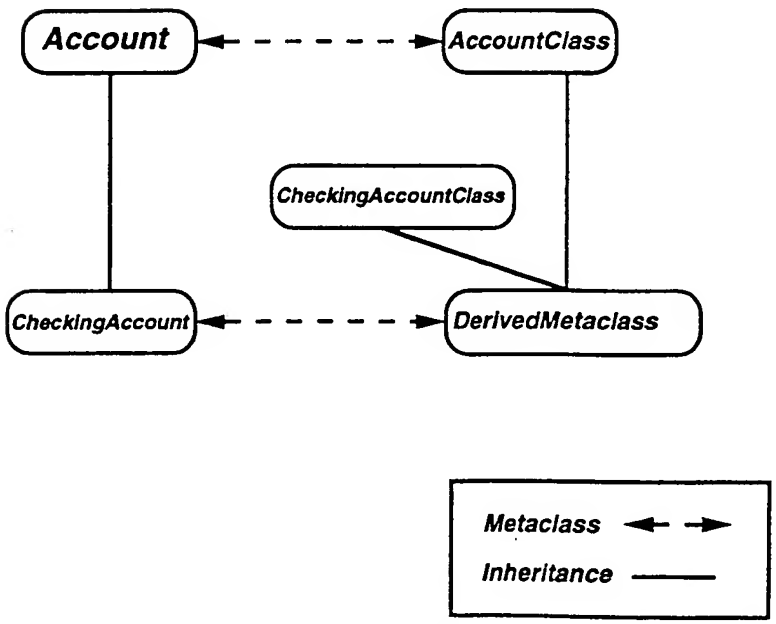


Figure 10.3 The solution: DerivedMetaclass.

10.1.4 Method resolution in SOM

SOM supports a very flexible notion of method resolution. In fact, three resolution strategies are supported:

- *Offset resolution.* This is the strategy used by C++ virtual functions. Polymorphism is implemented based upon the inheritance hierarchy and relying on static typing. To perform method resolution, a method token is retrieved from the class that first defined the method. This token is used as an index into the object's method table. Note that the names of the method and the class that first defined the method (in the superclass chain) must be known at compile time, since the method table access code is created at compile time.
- *Name-lookup resolution.* This is the strategy used by Smalltalk. Polymorphism is implemented using the object type and the methods it (or one of its superclasses) implements. Name lookup does not require the class name to be known at compile time. Instead, the search for the method is done at runtime. When an object receives a method invocation, the class object is used to determine which code will be executed.
- *Dispatch-function resolution.* This allows any dispatch strategy to be implemented by the object implementation. This strategy allows the class to provide any criteria for determining what code will be performed as a result of the method invocation. This strategy is similar to name lookup resolution in that the class object is consulted. However, the class is not restricted to a particular algorithm in terms of its response; it may use any lookup algorithm. Details of dispatch function resolution are beyond the scope of this chapter; the interested reader is referred to IBM (1993a).

The three method resolution strategies are ordered by decreasing performance and increasing flexibility. The default resolution strategy used in SOM in the programming language bindings available for SOM thus far (C and C++) is offset resolution.

10.1.5 SOM runtime objects

The SOM runtime environment includes a number of objects which are required for internal runtime support. These objects are automatically created when the environment is initialized:

```
SOMObject
SOMClass
SOMClassMgr
SOMClassMgrObject
```

The first three of these objects are class objects. `SOMObject` is the root class for all SOM classes and provides behavior common to all classes. It must be created to allow any class (and therefore any object) to be created. `SOMClass` plays the parallel role for metaclasses. Finally, `SOMClassMgrObject` is an instance of `SOMClassMgr` (and can therefore be created only after `SOMClassMgr` is). `SOMClassMgrObject` manages the classes used by the application and the environment, including the loading of the classes from the class libraries.

10.2 The Collection and Communication Classes

The SOMObjects toolkit provides a library of collection and iterator classes. These classes are similar to other commercially available library classes implementing various collections such as sets, lists, and dictionaries. The benefits for a SOMObjects user of using the SOMObjects collection classes over other such class libraries are twofold. First, these classes are already provided as part of the toolkit, relieving the user of the work (and the cost) required to integrate a new library into the environment. Second, the classes are SOM classes; multiple programming languages and environments may therefore make use of the collection classes. IDL definitions are provided for these classes to enable this language independence.

The collection classes are based on work done by Taligent (which is partly owned by IBM) and are primarily modeled as C++ collection classes. Figure 10.4 shows the hierarchy of the SOMObjects collection classes. An additional hierarchy provides classes for iterating over

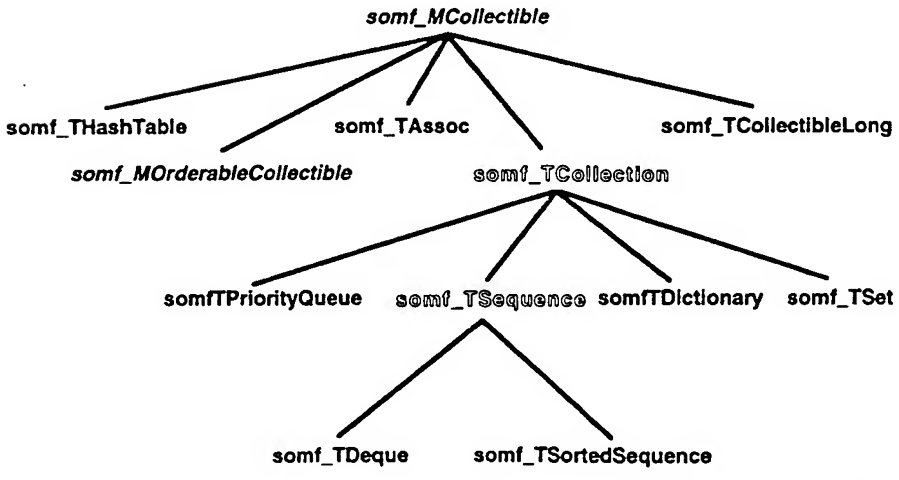


Figure 10.4 Hierarchy of SOMObjects collection classes.

collections. Like many collection class libraries, the SOMObjects collection classes are composed of abstract classes and concrete classes. The abstract classes in Fig. 10.4 (shown in outline) are `somf_TCollection` and `somf_TSequence`. The concrete classes provide data structures and operations implementing dictionaries, hash tables, linked lists, priority queues, queues, sets, sorted sequences, and stacks.

The `somf_MCollectible` and `somf_MOrderableCollectible` are *mixin* classes. These classes do not normally have semantics allowing them to be used by themselves; they are usually used in conjunction with other classes to produce new subclasses with specialized behavior (see Booch, 1991). The `somf_M` prefix is used in SOMObjects to denote mixin classes. The two mixin classes used in the collections hierarchy are used to provide operations necessary for containment in collections. This means that for an object to be contained in any of the collection classes' objects, its class must inherit from one of the collection mixin classes. By doing so, the object will supply necessary operations that will be used by the collection objects. For example, `somf_MCollectible` provides the `somfIsEqual` and `somfIsSame` operations, which are required for such collection operations as insertion, retrieval, and deletion (in most collection types).

In addition to the collection classes, the SOMObjects toolkit provides wrappers for socket functionality (Stevens, 1990). The `TCPIPSockets` class provides access to TCP/IP sockets, and the `IPXSockets` class provides access to Netware IPX/SPX sockets. The actual socket classes supplied with the SOMObjects toolkit support DSOM, the Replication Framework, and the Event Management Framework. Developers can create their own socket subclasses for their own specialized behavior. Note that the supplied classes provide the general framework for implementing such classes, but the concrete classes supplied which are used by DSOM, the Replication Framework, and the Event Management Framework should not be directly used by developers.

One of the advantages provided to a developer by these socket classes is that the socket interface is expressed in IDL (in the file `somssock.idl`) and can therefore be used in multiple programming languages and environments. Note that since most of the classes are intended to be used by internal frameworks of SOMObjects, only the IDL does not completely conform to CORBA IDL.

10.3 The Interface Repository Framework

The Interface Repository Framework is a set of classes providing programmatic access to the IR in SOMObjects. Applications may use

these classes to access the definitions stored in the IR. These classes implement the IR application programming interface (API) as defined by CORBA. The framework, together with the actual SOM IR database, provides a CORBA-compliant implementation of the IR and the IR interfaces. The SOM IR extends the CORBA definitions by storing additional information such as SOM modifiers and providing operations to access this information.

The actual SOM IR database is a set of flat files. Each IR file is constructed by running the SOM compiler with the IR emitter over a specific `.idl` file. The file `som.ir` is provided with the toolkit and contains the IR information for the built-in SOM classes.

The API supported by the SOM Interface Repository Framework conforms to the CORBA definitions as presented in Chap. 3. The main operations provided by the classes in the framework are summarized in Table 10.2. The structure of the IR follows the CORBA definitions. The SOM IR maintains objects of type `ModuleDef`, `InterfaceDef`, `AttributeDef`, `OperationDef`, and so on. The `Repository` object is the container through which access is initiated and is accessed using the `SOM_InterfaceRepository` macro, using `SOMClassMgrObject`'s `somGetInterfaceRepository` operation, or using the `RepositoryNew()` method. The SOM IR also provides full TypeCode support.

TABLE 10.2 SOM IR Operations

Operation	In Type	Description
contents	Container	Return a sequence of the contained IR objects.
describe_contents	Container	Return a sequence of ContainerDescription structures for the contained objects.
lookup_name	Container	Return a sequence of objects matching a name. The lookup recursively returns all such objects in the containment hierarchy under the target container object.
describe	Contained, XXXDef classes	Return a Description structure containing the IDL definition information.
within	Contained, XXXDef classes	Return a sequence of container objects which contain the target object.

10.4 Distributed SOM (DSOM)

SOM uses a CORBA IDL as the language for defining interfaces. This allows applications using SOM to be language-independent; classes written using SOM can be used for any language for which a SOM IDL binding exists. However, SOM does not address distribution issues; this is handled by DSOM. DSOM extends SOM by allowing applications to use distributed objects in a location-transparent manner in addition to the implementation transparency provided by SOM. Objects in distributed networks or in other address spaces on the local machine may be accessed and used by programs in a transparent manner. In fact, the actual location of the object will usually be hidden from the program using the object. SOM IDL is used by DSOM to ensure that the program using the object does not depend on the object implementation. The client is therefore decoupled from the object implementation following the decoupling principle of CORBA. While using the base SOM functionality provides programming language independence within a machine, DSOM adds the distribution capabilities requested by CORBA. DSOM therefore extends SOM to supply an implementation for a CORBA 1.1-compliant ORB that is used by applications using SOMObjects.

SOM and DSOM were designed to be CORBA 1.1-compliant, adding only functionalities that are not fully defined in CORBA, yet must be addressed by any real ORB implementation. For example, life-cycle services were not yet defined when SOMObjects came out as a commercial product. Since any commercial ORB implementation must address how objects are created, moved, copied, and deleted, DSOM provides its own API for remote object creation. Another example is the issue of ORB initialization, which will be defined only in CORBA 2.0. SOMObjects will evolve over time to use the specifications accepted by the OMG so that the SOMObjects toolkit will remain as close to the OMG specifications (in terms of both ORB functionality and Object Services) as possible.

Apart from such extensions, SOM and DSOM follow the CORBA definitions closely. Some examples of DSOM's implementation of the CORBA 1.1 concepts are:

- The DSOM class `SOMDObject` implements the CORBA 1.1 `Object` interface. `SOMDObject` objects implement the CORBA object reference concepts as proxy objects. The proxy represents the actual object (which may be on any local or remote machine) in the client's address space and supports the delivery of requests from the client to the object implementation. The client can therefore maintain a view where the location of the object is irrelevant, since the local proxy serves as the object from the client's point of view. Proxies are

created by DSOM whenever an object reference is passed; the client program need not be aware of this distribution issue. See Sec. 10.4.1 for more details.

- The implementation repository is used by DSOM as outlined in CORBA. `ImplementationDef` objects are used as defined by the CORBA `ImplementationDef` interface.
- The SOMOA DSOM class implements CORBA's BOA while providing a number of extensions. SOMOA uses the SOM compiler and the SOM runtime system to implement method dispatching. SOMOA also provides some of the ORB interfaces as defined by CORBA object adapters (e.g., `create`) as well as implementing the methods required for performing the mapping between object references and object implementation.
- DSOM supports the Context object as defined by CORBA.
- DSOM provides a full implementation of the CORBA DII.

10.4.1 DSOM proxies

`SOMDObject` objects implement CORBA 1.1 object references using proxy objects (see Fig. 10.5). When an object reference is passed to a client program, DSOM automatically creates the proxy object in the client's address space and makes the connection with the real object implementation using its own services. This automatic creation is done in a dynamic manner according to the object type. For example, if an object reference to an Account object is passed to a client program, DSOM will automatically build an `Account_Proxy` class that inherits from `Account` as well as from `SOMDClientProxy` (see Fig. 10.6).

`SOMDClientProxy` inherits from `SOMDObject` and provides the actual proxy support. In the construction of this new class (`Account_Proxy`), the SOM runtime system is used to dynamically create this class for use in the client's address space. `Account_Proxy` will include a method implementation for each `Account` method that

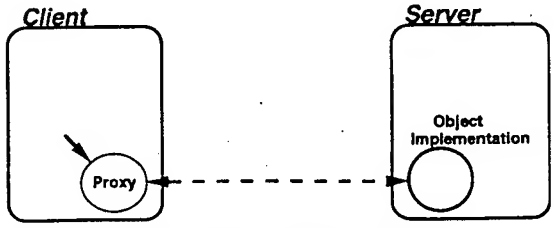


Figure 10.5 Proxy objects as reference objects.

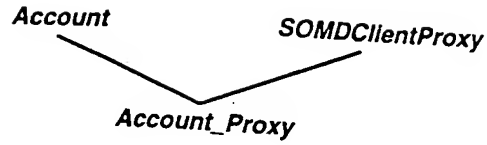


Figure 10.6 Automatic proxy class generation.

will simply delegate to the real object's method implementation using the DSOM remote invocation support.

10.4.2 DSOM client programming

DSOM client programming is similar to SOM client programming. Differences arise because the DSOM runtime environment must be used for remote life-cycle operations. These services are provided by the DSOM Object Manager `SOMDObjectManager` (derived from the `ObjectMgr`) class, which implements life-cycle operations, object location, and object activation operations. Developers may create their own specialized object manager by deriving a class from `ObjectMgr` and installing an instance of the new class as the object manager. `SOMDObjectMgr` provides basic life-cycle capabilities (creation and destruction), operations for locating servers implementing a certain type, and finding objects by id.

The steps taken by the client programmer in writing a DSOM client application are:

- *Initialization.* The DSOM runtime environment must be initialized (using `SOMD_Init`) before DSOM is actually used. All classes which will be used by the program must also be initialized by calling `XXXNewClass` (e.g., for the `Account` class, `AccountNewClass` would be called).
- Actual access of remote objects through the use of local proxies is then enabled. Objects are located or created using the DSOM runtime environment, as will be detailed below.
- Methods are invoked on remote objects through the local proxies. A proxy object appears to the client as the real object (in the local address space), and therefore requests to this actual remote object seem like no more than local SOM requests. This provides the DSOM distribution transparency.
- Object references passed as arguments and return values are automatically converted to proxies by the DSOM runtime environment completing the support for distribution transparency. Arguments and return values which are proxies are automatically converted to object references, so that when the object implementation receives the request, only standard object references appear.

- Objects that have already been created can be looked up by clients. Instead of creating a new object implementation whenever a service must be provided, the client may look for an already created object implementation.
- DSOM also supports externalization of proxy objects. It is possible to save references to remote objects by creating a string representation for any proxy and later using this external representation to reconstruct a usable proxy. The remote object must be identified and located. This is done by using the DSOM `somdGetIdFromObject` and `somdGetObjectFromId` operations.
- When the remote object is not used by the client, it may be destroyed. Operations are provided to release only the proxy, only the object implementation, or both.
- Finally, the client program will finalize the DSOM runtime system by calling `SOMD_Uninit` and any other SOM finalization procedure.

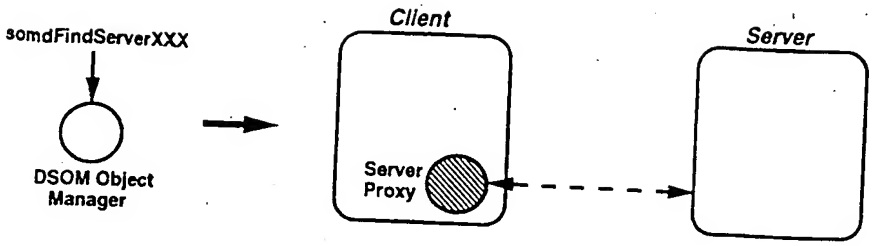
10.4.3 Creating remote objects

DSOM provides various methods for creating objects. These differ as to whether the client cares about which server will actually create (and contain) the object implementation. The simplest operation is `somdNewObject`. In this case, the client does not care about the actual server where the object implementation will be instantiated, and `SOMD_ObjectMgr` is free to use any server that it deems appropriate. If the client wants the object implementation to be created in a specific server, other creation methods should be used. For example, `somdFindServerByName` can be used to access a specific server. This operation creates a proxy object for the remote server on which the `somdCreateObj` operation may be used to actually create the object implementation in that server. Servers may be selected by name, by id, or by whether the server supports a class specified by the client; this last option provides a simple and partial Trading Service. Figure 10.7 shows the two phases of creating the server proxy and creating the object proxy.

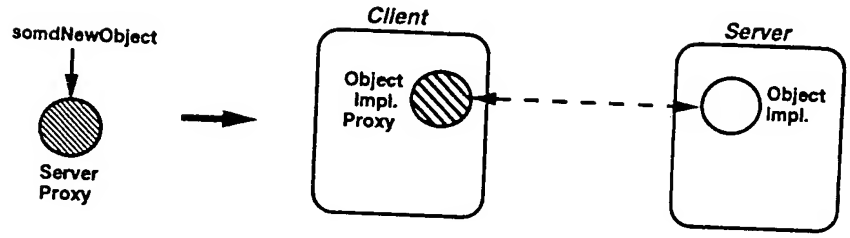
If the object is to be created using a specialized constructor, `somdNewObject` should not be used, since the default construction process, `somNew`, will be inappropriate. Instead, `somdGetClassObj` should be used to access the class object, which can then be used to invoke the specialized constructor.

10.4.4 DSOM server programming

Servers execute and manage object implementations in SOMObjects. The `SOMDServer` class provides instances of default server objects.



Phase 1 - Creating the Server Proxy



Phase 2 - Creating the Object Implementation Proxy

Figure 10.7 Creating the server proxy and the object proxy.

This class may be subclassed to provide specialized server behavior. The server object includes a SOMOA object—the SOM implementation for the Basic Object Adapter. Together these objects provide functionality for object activation, request dispatching, marshaling and unmarshaling, and so on.

For a server to be activated, its definition must be registered in the SOMObjects Implementation Repository as an `ImplementationDef` object. When a request to start a server is made through the somdd daemon (or during manual startup), the Implementation Repository is searched for this object. When it is found, the server may be started and the DSOM connection process continued; otherwise, service cannot be provided. The `ImplementationDef` object is maintained in the `SOMD_ImplDefObject` global variable and is used by the DSOM runtime system.

A server program usually includes the following steps:

- *Activation.* Server activation is done by the DSOM somdd daemon or by a manual process. The somdd daemon runs on every machine that provides DSOM support and is used as an "agreed-upon meeting place" for clients and servers. This is used by the DSOM runtime system to request service startup. Manual startup can be performed either by a command line request or by an application.
- *Initialization.* The server program must go through an initialization process that includes:
  1. DSOM runtime system initialization using `SOMD_Init`
  2. Initialization of the `ImplementationDef` object after retrieving it from the Implementation Repository
  3. Initialization of the object adapter using `SOMD_SOMOAObject = SOMOANew();`
  4. Application-specific initialization
  5. Invoking `_impl_is_ready` on the SOMOA object to inform the server that request processing may be initiated
- *Processing requests.* This is done either by passing control to a SOMOA loop (`_execute_request_loop`) or by maintaining control over the main loop within the application and calling `_execute_next_request` multiple times.
- *Exit.* By calling `_deactivate_impl` on the SOMOA object to finalize request processing and by calling finalization routines such as `SOMD_Uninit` and `SOM_UninitEnvironment`.

10.4.5 The DSOM management objects

Every server has a server object that functions as the server manager together with the object adapter. The default objects supplied by DSOM are the SOMDServer object and the SOMOA object.

The `SOMDServer` class provides methods for creating objects (`somdCreateObj`), deleting objects (`somdDeleteObj`), and finding class objects by id (`somdGetClassObj`). Methods are also provided for mapping between SOMObjects and SOMDObjects and for methods dispatch for SOM objects. These are used by SOMOA when mapping remote requests to local method invocations. When SOMOA is ready to dispatch the request, it calls `somdDispatchMethod` on the SOMDServer object, which will usually invoke `SOMObject::somDispatch`. The developer may override this default behavior.

If an object reference is required (for example, for a return value), SOMOA is used to create SOMDObject references using `create`, `create_construct`, or `create_SOM_ref`. Each of these has a

slightly different implementation and representation; these details are beyond the scope of this chapter (see IBM, 1993a).

10.5 The Replication Framework

The SOMObjects Replication Framework facilitates the creation of applications where multiple users share a common view and interact with one another based on this common data. Such applications are sometimes called "groupware" applications. Applications of this type are becoming more common in the marketplace as time goes on, since they provide increased possibilities for cooperation and efficient use of information systems. In such applications, users that are geographically distributed can share the common view, which makes it appear as though they were sitting in one room and using a whiteboard. Groupware applications may involve a common board of a computer game, a military scenario and plan presented on a map, or a design of an electric circuit board, among others. The users interact by changing their views and having these changes propagated to the other users' views. For example, this would allow one user participating in a circuit design session to propose a change in the design by manipulating the graphical representation of the design. This would be propagated to the other participants, who would see the change appear in their views. They could then comment on the change by making other alterations to the design.

The SOMObjects Replication Framework provides the basis for such cooperative applications. The framework allows an object to be replicated in multiple address spaces in such a way that each replica is aware of the possible presence of other replicas. A change to the object is propagated to all the other replicas. Propagation is independent of the number of replicas. In fact, the application does not know the number or the locations of the replicas. The replicas may even dynamically change; i.e., a replicated object may dynamically leave or join the group of replicas.

The Replication Framework is based on the Model-View-Controller (MVC) paradigm of Smalltalk (Goldberg and Robson, 1983). The object state serves as the model, while the different replicas are manipulated by the application views. The views and the model communicate using an agreed-upon protocol. The Replication Framework provides the necessary support for managing the model. The interfaces for viewing the model are handled by the application. This is similar to the semantic/presentation split paradigm offered by HP Distributed Smalltalk (see Chap. 9).

To use the Replication Framework, the developer will inherit from the `SOMRReplicbl` class. For example, if an ElectronicCircuitDesign object is to be replicated, then a new class will be created by the

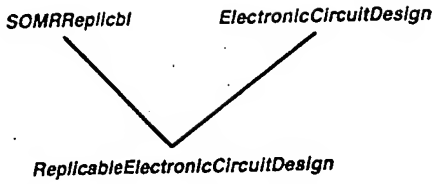


Figure 10.8 Example class for object replication.

developer, as shown in Fig. 10.8. By inheriting from `SOMRReplicbl`, the new class will support replication semantics. Support for the Replication Framework is also provided by the `SOMR` class. An instance of this class must therefore be created if the Replication Framework is used.

The Replication Framework uses socket type communication for propagating the changes between the different replicas. To do this, the framework must agree on the ports on which messages are sent. The Replication Framework uses a file, called the `.scf` file, specifying these attributes. This file is used only for the initialization process. Once the communication has been set up, the framework does not use any files, allowing it to provide good performance.

The Replication Framework uses a master-slave protocol. Among the replicas, one of the objects is the master. All propagations are actually done by that one object. This means that when any object desires to make a change, it will inform the master, and the master will then propagate the change. During the change, no other replica may ask for a change (this is similar to a lock being formed by the master). The number of messages for a single update for N replicas is therefore either $N - 1$ (if the master performed the update) or $N + 2$ (one message to effectively obtain the lock, one to send the update to the master, $N - 1$ messages for the update, and one to release the lock). The updates will always be serialized, since the master provides the "monitor segment." In addition, the master will always send the messages to the replicas in the same order, so that no matter which replica triggered the change, the order in which the replicas receive the change remains the same. The Replication Framework includes algorithms supporting fault tolerance that will elect a new master if the master replica should go down.

10.5.1 Operation and value propagation

The Replication Framework supports two propagation models. In operation propagation, the actual method which causes the change is propagated. If one of the replicas is modified as a result of invoking a method on it, then the method specification is propagated to the other

replicas and invoked for all the other replicas. Note that for the replicas to achieve the same state, some limitations on the method must be observed. For example, if the method uses the physical machine's clock as a part of the state change computation, then the changes of the different replicas may differ. In value propagation, on the other hand, the actual state changes of the object are propagated to the replicas.

The two different propagation models differ in what is being propagated. Choosing the best propagation method is important for the application's performance. Each of the models involves a tradeoff between propagation time and the time for producing the state change. Each of the two models has characteristics making it more suitable in certain cases. As a rule of thumb, if the update methods are easily computed and cause large change sets, then operation propagation will be more efficient. If the change methods are computationally intensive (or otherwise resource intensive) but produce small change sets, then value propagation is preferable. In cases where the tradeoff is less clear, the selection should be based upon application benchmarks. Figure 10.9 shows the tradeoff space.

The actual decision on the propagation made is done by the method which updates the object. If operation propagation is used, then the `_somrLockNlogOp` and `_somrReleaseNPropagate` operations will be used to delimit the method which performs the update. If value propagation is used, then the `_somrLock` and `_somrRelease PropagateUpdate` calls delimit the updating method.

10.5.2 Directives

Since the objects participating in the Replication Framework are distributed on different machines, it is possible that events will affect

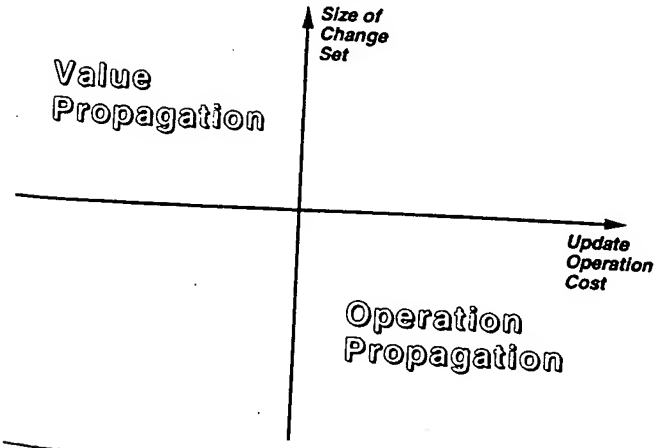


Figure 10.9 Tradeoff between value and operation propagation.

TABLE 10.3 SOM Replication Framework Events

Directive	Description
LOST_CONNECTION	Connection with other replicas has been lost but attempts to reestablish it are being made; no updates to the replica should be attempted.
CONNECTION_REESTABLISHE D	Connection to the other replicas has been reestablished.
BECOME_STAND_ALONE	The Replication Framework has given up trying to reconnect to the other replicas.
LOST_RECOVERABILITY	The .scf file cannot be updated.

the connectivity and availability of the different replicas. The actual propagation management is done by the Replication Framework classes, yet the participating objects must have a way of being informed of changes to the connection topology. This is accomplished through the use of directives. Directives are methods that are sent from the Replication Framework classes to the applications using the replicas and inform them of events such as network faults. Examples of such events are shown in Table 10.3. To accept directives and provide a response, the application should override the `somrDoDirective` method which is called by the Replication Framework classes with an argument specifying the directive string.

10.6 The Event Management Framework

The Event Management Framework provides support for registering interest in events and receiving notifications when an event occurs. It is similar to event handling in most windowing systems and other event managers (e.g., Teknekron, 1994). Using the event manager in SOMObjects involves registering interest in the event (by providing a callback specification to be invoked when the event occurs) and passing control to a central framework function that never returns. The program will invariantly remain in that function until one of the events takes place. At this time, the callback function will be invoked. When the callback function terminates, control is passed back to the main loop function; this is depicted in Fig. 10.10.

The Event Management Framework is extensively used by DSOM and by the Replication Framework. Any interactive applications

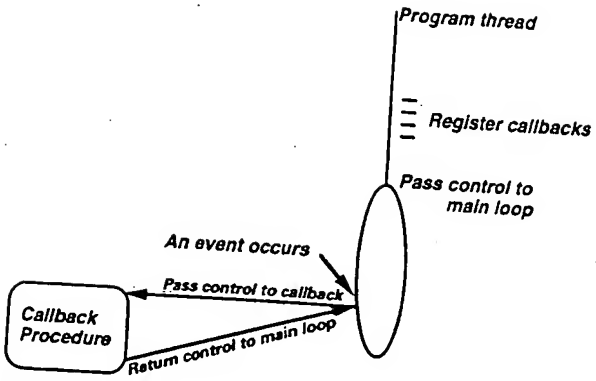


Figure 10.10 Passing control to the main loop and callbacks.

requiring event handling and using DSOM or the Replication Framework must therefore use the Event Management Framework (so that event handlers will not clash).

The Event Management Framework is especially used in single-threaded systems. In multithreaded systems, where it is possible to spawn multiple threads in any application process, event management becomes much simpler. If an application is required to respond to multiple events, for example, separate threads can be created, each with a responsibility toward a different event. This design relies on the possible usage of multiple threads; if only a single-thread model is supported, event management must be handled by a central component which reacts to all events and delegates the work to different callback functions. However, even in multithreaded systems, the Event Manager Framework is useful. Multithreaded environments must ensure thread safety. Scenarios in which multiple threads may require access to shared resources must be managed. The event manager itself in SOMObjects is guaranteed to be thread safe; its consistency is guaranteed even in the presence of concurrent operations. The event manager interactions are guaranteed to produce results identical to a serial ordering of all interactions. This is an important feature which can be used even when multiple threads are involved.

The Event Management Framework is defined in IDL, so that access can be provided for multiple programming languages as long as the IDL binding is available. Such bindings are easily created using the Emitter Framework.

7 The Emitter Framework

As is appropriate for a CORBA-compliant implementation, the heart of SOM is the IDL interface definitions. All interfaces are written in

IDL. The SOM compiler is then used to produce a translation of these interfaces to the native programming language or programming environment in which the implementation is actually done. SOM is intended to support many programming languages and to produce a common framework to be used by many programmers, using many programming languages, on many operating systems, running on many hardware platforms. For each such programming language that will be used with SOM, an IDL compiler must be produced; this will enable translating the IDL definitions to that programming language. To facilitate and ease the production of such translators, the SOMObjects toolkit includes the Emitter Framework.

The Emitter Framework is designed to allow developers to write extensions to the SOM compiler. Such extensions can be used to write any translator for the IDL specifications. These translators take the form of "emitters" which emit translated components derived from the IDL specifications and translation rules and formats. These emitters can then be embedded in the IDL compiler to allow for the new translation. This is mainly useful for programming language bindings for IDL, but it is also very useful for other purposes, such as generating automatic documentation from the interface definitions and deriving component information which can be used by CASE tools.

Figure 10.11 illustrates the function of the Emitter Framework components. The IDL interfaces and definitions are read by the IDL parser to produce an Abstract Syntax Graph (ASG). This graph repre-

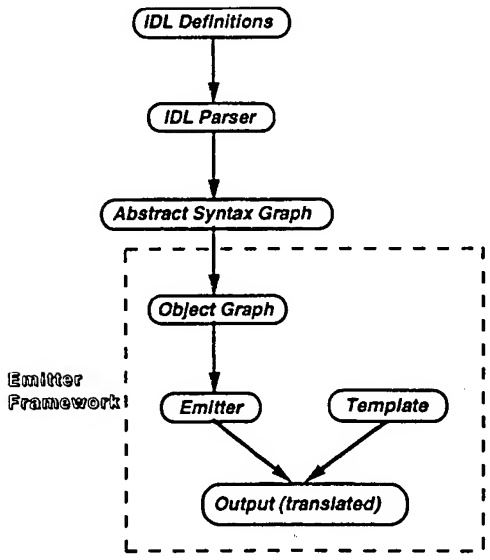


Figure 10.11 The Emitter Framework components.

sents the IDL constructs, such as operations, parameters, attributes, and interfaces. This part of the SOM compiler is identical for all translators and helps decouple the various translators from dependence on specific IDL syntax. All translators are based on the same IDL syntax, so this decoupling places no limitations on translators. It assures that if the IDL syntax were to be changed, only the IDL parser would be affected and the large base of translator code would not need modifications (so long as the new semantic constructs supported the semantic constructs used by developers).

Once the ASG has been created, it is used to form an object graph that represents each element in the ASG as an object. Each such object is an instance of one of the built-in `SOMTXXXEntryC` classes, where XXX is replaced by the various syntactic elements (e.g., `SOMTParameterEntryC`). The programmer may replace one of these built-in classes by a specialization allowing new object types to populate the object graph and allowing for an additional level of extensibility and flexibility.

Once the object graph has been created, the emitter begins the translation process from the object graph structures to the desired output format. This process uses a template for the output format which specifies the location and format of the various structural components. Since most of the formatting specifications are part of the template and not part of the emitter, it is very easy to modify the formatting rules of the translation.

The emitter is an instance of the built-in `SOMTEmitC` class or one of its (direct or indirect) subclasses. When a programmer extends the SOM compiler with a new translator, a new class in the `SOMTEmitC` hierarchy will typically be created. In addition, a template file will be created. The new emitter class will inherit most of its functionality from `SOMTEmitC` and override some of the translation generation methods. The template is used for defining location and formatting rules for the generated output.

10.7.1 Emitters, templates, and entries

The Emitter Framework includes three categories of entities which participate in the translation process. The emitter itself is the process manager, the template defines the output format, and the entity objects represent sections of the IDL specifications and are used as entry points with which translation behaviors can be associated. The emitter uses the information maintained in the entry objects, and uses the template for output creation. Each of these entities is implemented by a class in the Emitter Framework: the emitter class `SOMTEmitC`, the template class `SOMTTemplateOutputC`, and the class hierarchy rooted at `SOMTEntryC`.

TABLE 10.4 SOM Emitter Framework IDL Default Sections

prologS	Information that is emitted before any other.
attributeS	Class attribute information.
methodS	Class method information. Is invoked iteratively for every method in the class.
interfaceS	Information for the interface.
moduleS	Module information.
baseS	Information about the parent classes; used when a class inherits from others.
epilogS	Information emitted after all others.

Creating a new emitter typically involves two things: the emitter and the template. First a subclass of `SOMTEmitC` is created to provide specialized control of the translation process. The `somtGenerate Sections` is the method that is most commonly overridden to provide the customized specification determining what is emitted and in what order. This is done by attaching emitting methods for standard sections of the IDL. Some of the default sections are given in Table 10.4. Each of these sections has a method which is used to emit the section. By overriding the default definitions of these methods, customized emitter behavior can be defined.

Once the emitter class has been defined, the output is created by rendering the syntax graph on the template definition. The template defines the output format, locations, and symbols which will make up the resulting files. Template creation is a simple yet very powerful process. It is very easy to create emitters in a matter of minutes or hours for more complex translations, since much of the specification goes into the template definitions.

10.8 The Persistence Framework

The SOMObjects Persistence Framework provides persistence support for SOM objects. The Persistence Framework allows SOM objects to be stored in files, which allows the objects to exist after program termination. The objects can later be reconstructed to the internal representation. In many respects the Persistence Framework is really an externalization/internalization framework. However, the Persistence Framework is not limited to using external files; it can be

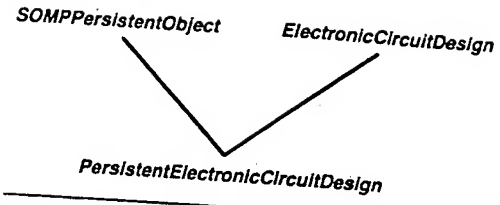


Figure 10.12 Creating a class for persistent objects

extended to use other storage strategies. The Persistence Framework provides both a base service that can be immediately used by developers as is to provide simple persistence and a complex framework that can be extended to support specialized persistence mechanisms. Customization can include customized object clustering, specialized persistence formats, and the use of specialized repositories. The basic service uses ASCII and binary files, simple object clustering, and externalization type formats.

To allow a SOM object to be saved using the Persistence Framework, the object's class must inherit from the `SOMPPersistentObject` class, (see, for example, Fig. 10.12). A persistent object must support the saving of its state to the persistent store. If the developer does not wish to provide extra code for making the object persistent, then the class must conform to the storing methods supplied by the default framework classes. The default classes provide generic facilities and do not handle special data elements. The storing operations provided by the default classes therefore handle only the CORBA-defined data types (e.g., short, long, array, union, double, etc.). Thus, any object that desires to be made persistent must be composed only of CORBA data types; otherwise specialized storing and retrieving methods must be provided. The default storing methods are similar in style to the `write_*` methods used in the Externalization Service discussed in Chap. 6. The default class used for storing object states comprising of CORBA data types only is `SOMPAttrEncoderDecoder`.

10.8.1 Persistent object ids and groups

Persistent objects are assigned persistent object IDs (OIDs). The persistent OID must be associated with an object before it can be externalized or internalized. This OID uniquely identifies a persistent object. The ID string value not only identifies the object but also embeds information about how the object should be stored and where. This string has the format of

<group manager class name>:<group name>:<group offset>

The group manager class name defines what class will be used to store the object. The group name defines where the object will be stored, and the offset specifies where precisely in the group the object will be stored, since multiple objects will typically be stored in a single group. For example, the default group manager class is `SOMPAscii`, so a possible persistent OID might be

```
SOMPAscii:myfilename:0
```

where `myfilename` is the name of a file. The `SOMPAscii` manager uses ASCII files for the persistence of objects; `myfilename` would therefore be the file in which the object would be stored at offset zero.

Persistent OIDs may be associated with an object in one of three ways:

1. An OID can be explicitly created (by creating and populating a `SOMPPersistentId` object) and associated with the object using `sompInitGivenId`.
2. A `SOMPAssigner` object can be used to create an OID (using `sompInitNextAvail`) and associate it with the object.
3. The developer may request that the object be placed near another object and the OID derived accordingly (a simple form of object clustering).

The last method is the one which is most commonly used. Typically only one object will be explicitly given an OID and the rest of the objects will be given OIDs by placing then near a previously identified object.

Groups are the unit of object storage and are collections of objects stored together in a single file (or some other persistent organization). Groups are represented in the Persistence Framework as instances of `SOMPIOGroup` and are associated with a group manager object (whose class derives from `SOMIOGroupMgrAbstract`). The group manager defines how the objects are stored, whereas the group specifies where the objects are stored.

The SOMObjects Persistence Framework supplies two built-in group managers: `SOMPAscii` and `SOMPBinary`. `SOMPAscii` is the default group manager; it stores each group as a single ASCII file. The `SOMPBinary` group manager is similar, but it stores the data in a binary format (e.g., the number 17 will not be stored as two characters "1" and "7" but as the binary representation of 17). Both the `SOMPAscii` and the `SOMPBinary` group managers have similar characteristics. The `SOMPAscii` group manager is usually used at the initial development stages, since it is easier to debug (since the storage can be viewed in a text editor), and is later replaced by the `SOMPBinary` group manager, since it is more efficient to use.

10.8.2 Storage formats

The file containing the persistent representation of a group of objects is a combination of persistent object information and group specification information. These are coupled together throughout the file; prior to the information for the actual object externalization, the object's grouping information is described. These sections have a format which is different from that of the sections holding the persistent object information. The format of the persistent object is called the Persistent Object Format (POF), whereas the format of the group information is called the I/O Group Format (IOGF).

Every persistent object is associated with an Encoder/Decoder object. The class of the Encoder/Decoder determines what POF will be used for storing the object. The Encoder/Decoder class is where the actual read and write methods are implemented and is responsible for the object externalization/internalization. The association of a persistent object with an Encoder/Decoder can dynamically change over the lifetime of the object. Clearly, when the object is restored, the same Encoder/Decoder class must be used as when it was stored. The association with the Encoder/Decoder is at the object level; therefore, different instances of the same class may still be associated with different Encoder/Decoder objects. It is also possible to associate a class with an Encoder/Decoder.

The SOMObject Persistence Framework provides a default Encoder/Decoder called `SOMPAttrEncoderDecoder`. `SOMPAttrEncoderDecoder` requires that every persistent element in the object be an attribute, that the accessor methods of `get` and `set` be defined, and that the attribute be declared with the `persistent` modifier in the IDL file. `SOMPAttrEncoderDecoder` uses a very simplistic POF that may be sufficient for many applications. For example, an object of type `Security` having attributes for the security's symbol, its name, the stock exchange name, and a price may be stored as

```
(4) (6) symbol (3) OBJ (4) name (14) Object Corp (14) stock exchange (4) NYSE (5)
price (6) 43.125
```

The first number indicates how many attributes are stored. The remainder is a list of pairs, each pair comprising the number of characters in the string and the string. Each attribute is represented as two of these pairs; one specifying the attribute name and one specifying the value.

In many cases the default Encoder/Decoder will be sufficient, but in some cases the developer must create a customized Encoder/Decoder. The developer would do this by creating a subclass of `SOMPAttrEncoderDecoderAbstract` and overriding the `sompEDWrite` and `sompEDRead` methods. To associate the newly created Encoder/

Decoder, the developer would use the `sompSetEncoderDecoderName` method to associate it with a single object or the `sompSetClassLevelEncoderDecoderName` if the Encoder/Decoder is to be associated with a class of objects.

10.8.3 Saving and restoring

The saving and restoring process uses the framework components as described above. The steps performed when storing (externalizing) an object are:

- Create a Persistent Storage Manager (PSM) by using the `SOMPPersistentStorageMgrNew` method to instantiate the `SOMPPersistentStorageMgr` class. This initializes the Persistence Framework.
- Create the object to be stored.
- Assign a persistent OID to the object in one of the three ways described above.
- Use the `sompStoreObject` method to store the object based on the information in the persistent OID.

To restore the object, the following steps are followed:

- Initialize the Persistence Framework as above.
- Create a new persistent OID object using `SOMPPersistentIdNew`; this is not a persistent object but a persistent OID.
- Assign the actual OID to the persistent OID object using `somutSetIdString`.
- Use `sompRestoreObject` to restore the object.
- Use `somFree` to free the persistent OID object.

The saving and restoring process also incorporates a passivation/activation process. Just before storing the object, the Persistence Framework calls the object's `sompPassivate` methods, and just after restoring the object, the `sompActivate` method will be invoked for that object. These methods are defined in `SOMPPersistentObject` but should be overridden by objects which have to supply specialized activation or passivation behavior.

Once the object has been stored, the PSM provides various management capabilities. For example, the `sompObjectExists` method can be used to determine whether an object with a certain persistent OID exists. An object can be deleted from its persistent store using the

`sompDeleteObject` method. The storage of the deleted object is not reclaimed unless one of the objects in the group is marked using `sompMarkForCompaction`. In this case, the next time the marked object is stored, a garbage collection process will compact the entire group (file).

AUTONOMOUS AND MOBILE AGENTS IN DISTRIBUTED NETWORK MANAGEMENT AND MONITORING SYSTEM

UMA SHANKER, Innovative Network Applications, IBM Scientific Center, 18 Vangerowstrasse, Heidelberg 69115, Germany Email : uma@de.ibm.com , Tel : ++49-6221-594560, Fax : ++49-6221-593300

Abstract Distributed object technology provides optimum architecture for Internet and intranet web applications. This paper presents a generic architecture solution for the Network Management System. Central to the distributed network management is the broker which provides access to the real-time information from the network nodes. It explains how the broker is used to manage and access management data on real-time basis. Finally we talk about the autonomous and mobile agents in the given architecture. It explains use of design patterns like broker, proxy etc. and some details and experiences from the ongoing project are described. Areas for future research are also explored.

Keywords : Agents, Network Management, **CORBA**, Java, SNMP

1 INTRODUCTION In todays growing internet its very inportant to have upto-date information about your systems components and it becomes more important if you are internet provider. Increasing size and complexity of network and the need to support all sorts of advanced services in a reliable and cost-effective manner pose a major challenge to network management.

The traditional approach to develop large complicated network mangement/monitoring systems and applications was to build them using a uniform platform-specific architecture, based on openly defined management interface such as defined by SNMP[6] management framework. In the traditional model, a network management application's user interface is provided by an

appropriate data-driven protocol to a windowing or other display(like in SunNet Manager[2] from Sun). Recent advances in technology such as WWW[3] browsers and services, corporate intranet, and executable contents such as provided by Java[7], introduce the possibility for a more network-centric approach to the development of management applications, that allows organisations to leverage their existing Network Management System (NMS) solutions with generic, highly scalable and customizable Web-based front end. Network management is data-based. Vast amount of information(especially in large, complex networks) are collected by the network agents[1] and sent to the manager site. Manager site collects network performance, status and configuration information, maintains historical and statistical data, handles events and reports. All this information, which explodes in size with network complexity and size augmentation, need not only be stored efficienty but it must be enriched with powerful data management features that allow the relization of demanding, high level management functions like temporal reasoning, decision-making, planning etc. Additional functionalty is also required in large multiple domain network environments. As we come into Web-based front end system, there are several places which are very critical to the performance and reliablity of the system. Very first point is, how to handle data in the reliable and easy way? Security is yet another important issue. We propose the

Distributed Network Management and Monitoring System(DNMMS) architecture, which is based on Java[7] and Object Request Broker[11]. We are using Voyager from objectspace, as in addition to the fundamental ORB features it also support mobile objects, autonomous agents, events and listeners, database independent persistence, directory services etc. On the other side, we are using Java due to its ability to load classes into a virtual machine at run time. This capability enables infrastructures to use mobile objects and autonomous agents as another tool for building distributed systems. DNMMS architecture is shown in figure 1.

2 BACKGROUND Various Tools/Systems are available today in the market for System Management and Monitoring. Please note that we are using the term System management and not Network Management, as in todays distributed environment, Internet providers are not only interested in there network components like routers, but also in there web-applications like http and proxy servers and they like to have common interface for all the management applications. Generally these tools runs on different Platforms and generally not

Figure 1: Architecture of DNMMS compartible with each other. Extension/upgrading is also very difficult. Now a days one of the common way to provide the statistics about the systems like web, routers, nodes, etc. using web-front end is to collect the statistics information at regular time interval at central location, where the data is formatted(mainly by CGI) for the administrator to display the data in the web browser. But CGI is a state-less technology, so we don't have any persistent state during the browsing. In general all of the data, some of them are never used by any body, is transmitted from the nodes to the central location. There is no connection with the system once data is retrived. If we want to have further special query about some node/machine, we have to use some other system or all the queries are done at the central location but not at the node/machine, which makes that tool not real-time and hence not 100% reliable which is necessary for the network components(routers/proxy/web servers) of the Internet Service Providers. Next section explains architecture of the Java/**Corba** based DNMMS system and also the extension and integration of existing tools.

3 OVERVIEW OF DNMMS 3.1 DNMMS Architecture Central to the DNMMS architecture are two components CountryBase and Embassy. CountryBase is implemeted according to the Broker Pattern[10].

On one side it collects management information from the network nodes and provides facilities for the real-time management data, on the other side it is used by the front end to present management data in the visual (graphical/text) format. Please note that frond end can be application or applet, which can access CountryBase by DNMMS API.

Embassy is based on the Proxy Pattern. It is situated at network nodes which provides data from that node according to the standard interface. Interface to the Embassy is described in the IDL format, which allows its implemetation in different programming languages like Java, C++. As Embassy will be running on different platforms, it allows to have optimal implementation for that platform. It gives possibilities to integrate existing management systems. Each network node can have several management objects. Every management object is described by its mangement object identifier(MOI). CountryBase requests management data from Embassy by providing MOI. MOI completely specifies manageable resource inside the DNMMS.

You can compare the DNMMS as a country, who has there embassies located at other foreign countries (different servers, routers). Most important is that Embassies are not engaged with the foreign countries government. They just provide the required information within the protocol set by the two countries. Inside DNMMS every object like embassy, agent, etc. have unique identification numbers called GUID (16-byte globally unique identifier).

3.2 CountryBase overview CountryBase is based on the Broker Pattern[8]. It is responsible for coordinating communication, such as forwarding requests, database and directory services as well as for transmitting results and exceptions. Instead of focusing on low-level inter-process communication, it allows to access distributed

services simply by sending message calls to the appropiate Embassy. In addition, the CountryBase architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of MOI. CountryBase just transmits requests made by the client to the appropiate Embassy. In general CountryBase is responsible for :

ffl Registration of embassy ffl Transfers of messages ffl Error recovery ffl Interoperation with other brokers

ffl Locating embassies ffl Directory services ffl Agent management ffl Storage of management data

CountryBase has a central distributed database which stores all the management data collected from the different embassies and also data to manage CountryBase itself. Every information exchange between a client and Embassy passes through the broker.

3.3 Embassy Overview Embassy is based on the Proxy pattern[9]. It is installed at every network node and it provides data from that node according to the standard interface. Embassy during the initialization process registers himself with the CountryBase. Embassy can be started through remote machines as all the classes needed for the Embassy can be automatically downloaded. This allows easy installation and management.

Client make request by providing MOI to the Embassy. After receiving request Embassy works according to the local implementation and provides back the result. General format of MOI is MOI://PROTOCOL/VERSION/OI/ where PROTOCOL is a name of protocol like SNMP, VERSION is a version of the MOI implmentation and OI is a Object Identifier, eg. in case of SNMP OI can be 1.3.6.1.2.1.7.1 for the UdpInDatagrams. Please note that this format on the one side allows to use industry standard like SNMP but are not only limited to that. All informations related to MOI are available through the CountryBase Directory Service under [/CountraNet/MOI]. Example of MOI is : MOI://SNMP/1.0/1.3.6.1.2.1.7.1/.

Interface to the Embassy is described in the IDL[5] format, which allows its implemetation in different programming languages like Java, C++. As Embassies may will be running on different platforms, so there implmentation can be different. Working of DNMMS using ORB and MOI is shown in figure 2.

4 AGENTS IN DNMMS 4.1 Agents in CountryBase DNMMS Agents are the voyager's mobile objects, they can move at run time from one virtual machine to another. In this way, agents can act independently on the behalf of a client, even if the client is disconnected or unavailable. CountryBase has central directory to hold every neccessary information to operate Agents in the DNMMS. Each Agent in the DNMMS is associated with unique identification number called AUID, which is

Figure 2: ORB and MOI working in DNMMS a GIUD of that Agents Virtual Reference[4]. All the informations are stored in central directory on CountryBase in [/Agent]. As the agent proceeds from embassy to embassy agent related data are stored in this central directory. This provide one point management of all the agents in DNMMS. This provides facilities to check status of different agents in DNMMS. Further enhancement can be done by providing agents related information on date, status, last logged, etc. basis. CountryBase is a central location to execute commands like start, stop, wait etc. on the agents.

4.2 DNMMS System Agents There are serveral DNMMS agents which are started at the CountryBase creation time. These agents are DNMMS System Agents and are generally live forever status[4]. They are used to manage DNMMS itself.

4.3 DNMMS Management Agents DNMMS management agents are the agents which can be used to monitor status of the network nodes. These agents are completely autonomous. It means that they can control themselfs in different situations. An agent is a special object type. An agent has autonomy. An autonomous object can be programmed to satisfy one or more goals, even if the object moves and lost contact with his creator. Using

voyager DNMMS supports autonomous and mobile agents, mobility is the ability to move independently from one device to another on a network. When agent moves to new location , he leaves behind a forwarder to forward messages.

If the agent want to have high-speed conversation with remote object, the agent can move to the object and then send it local Java messages.

Autonomous agent in DNMMS is useful for many reasons, for example:

ffl If a task must be performed independently of the computer that launches

the task, a mobile agent can be created to perform this task. Once created, the agent can move into the network and complete the task in a remote program.

ffl If the periodic monitoring of a remote object(in our case object return

back the status of himself at the CountryBase) is required, creating an agent that meets the remote object and monitors it locally is more efficient than monitoring the device across the network.

Agent knows nothing about the remote location address and other useful informations which is needed to access that location. Agent gets all these information from the CountryBase Directory Service. Once agent has virtual reference of the remote embassy he can use it to find out other informations like, GUID, network address, port and so on. As a example, there can be agent which goes to the target Embassy and tries to monitor the tcp segments received and send. If it increases 2000 segments/sec then agent reports to the output. Please note that any other action can be taken on this event. Agent can finally move to the CountryBase or can be parked at the Embassy or can send some notification and so on.

5 CONCLUDING REMARKS The growing complexity of the network infrastructure requires a reliable and real-time management system. A web-based, network centric design gives the flexibilty to offer such a solution, and additionally allows one to scale up the solution because of its genericity. We believe that architecture presented is one of the first such application of its kind. But there are still some important open issues like security, performance and workload.

6 ACKNOWLEDGMENTS Thanks should go to Dr. B. Lamparter for interesting comments and to Dr. J. Rueckert, Prof. P. Schmidt and Prof. W-D. Oberhoff for their encouragement.

References

[1] IBM's Intelligent Agents. http://www.networking.ibm.com/iag/iaghome.html. [2] Sun Net Manager. http://www.sun.com/solstice/index.html. [3] The World Wide Web Consortium. http://www.w3.org. [4] Voyager, The Agent ORB for Java, Core Technology User Guide Version

2.0 Beta 1. http://www.objectspace.com/voyager, 1997.

[5] **CORBA 2.2/IIOP Specification**. http://www.omg.org/**corba**/c2indx.htm,

Feb 1998.

[6] M. Schoffstall J. Case, M. Fedor. A Simple Network Management Protocol. http://ds.internic.net/rfc/rfc1157.txt, May 1990.

[7] Guy Steele James Gosling, Bill Joy. The Java Language **Specification**.

The Java Series. Addison Wesley Longman, 1996.

[8] Douglas C.Schmidt James O.Coplien. Pattern Languages of Program

Design. Addison Wesley Publishing Company, 1995.

[9] Norman L.Kert John M.Vlissides, James O.Coplien. Pattern Languages

of Program Design 2. Addison Wesley Publishing Company, 1996.

[10] Wolfgang pree. Design Pattern for Object-Oriented Software Development. Addison Wesley Publishing Company, 1995.

[11] Douglas C. Schmidt. Overview of **CORBA**.

http://www.cs.wustl.edu/ schmidt/**corba**-overview.html, Nov. 1997.